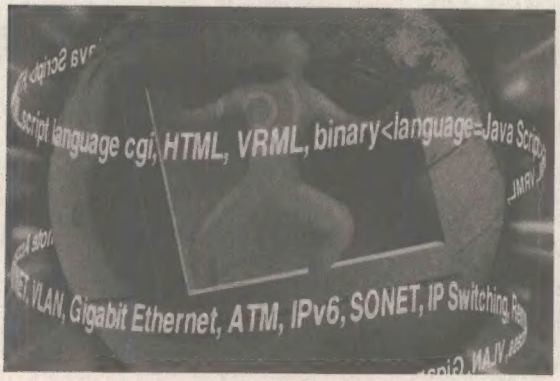
Delphi

ประพนธ์ อัศวภาณุวัฒน์

Debugging & Exception Handling

เน้นเรื่องการพัฒนาโปรแกรมให้มีเสถียรภาพ และการใช้เครื่องมือที่สามารถช่วยเหลือในการพัฒนาโปรแกรมให้รวดเร็วยิ่งขึ้น



สวัสดีครับ ในการพัฒนาโปรแกรมต่าง ๆ ของเรา สิ่งที่มีความสำคัญและไม่สามารถมองข้ามได้คือ การตรวจสอบความผิดพลาดของการทำงานในโปรแกรม (Debug) และการควบคุมความผิดพลาดที่อาจเกิดขึ้นจากการทำงานของโปรแกรม (Exception Handler) ทั้งสองเป็นสิ่งสำคัญที่ช่วยเพิ่มความรวดเร็วในการพัฒนาและเพิ่มเสถียรภาพให้กับโปรแกรม ซึ่งเป็นหัวข้อที่เราจะคุยกันในฉบับนี้ครับ

BUG

"บัก" (Bug) เป็นความผิดพลาดของโปรแกรมที่เกิดขึ้นในช่วง Run-time เกิดจากการทำงานผิดพลาดของลอจิกในระบบ มีผลทำให้ได้ผลลัพธ์ผิดจากสิ่งที่ควรจะได้รับจริง ซึ่งคอมไพเลอร์ (Complier) ไม่สามารถตรวจสอบพบได้

เป็นที่แน่นอนครับ ส่วนของงานพัฒนาโปรแกรมที่น่าเบื่อหน่ายที่สุดคือการค้นหาจุดที่เกิด "ความผิดพลาด" ภายในโปรแกรม ยิ่งถ้าเป็นโปรแกรมใหญ่ ๆ ที่มีฟังก์ชันการทำงานซับซ้อน หรือโปรแกรมที่เราไม่ได้พัฒนาขึ้นเองแล้ว ยิ่งเป็นเรื่องที่ไม่อยากพบพานเลย แต่ในความเป็นจริงแล้ว การพัฒนาโปรแกรมให้สมบูรณ์ร้อยเปอร์เซ็นต์ในครั้งเดียวคงเป็นไปได้ยาก และถึงแม้โปรแกรมจะสมบูรณ์ แต่โอกาสที่จะพบการทำงานที่ผิดพลาดของโปรแกรมก็ยังคงมี ซึ่งเป็นผลกระทบอันสืบเนื่องมาจากสภาพแวดล้อมเปิด (Open - Environment) ของระบบปฏิบัติการ ไม่ว่าจะเป็นความไม่ได้มาตรฐานของอุปกรณ์ต่าง ๆ ที่ใช้ในระบบ หรือความต้องการทรัพยากรของโปรแกรมในสภาพแวดล้อมต่าง ๆ กัน

แก้ไขที่ต้นเหตุ

"ไม่มีวิธีการแก้ไขปัญหาใด ๆ ดีเท่ากับการเตรียมพร้อมที่จะพบกับปัญหา" ที่ผมกล่าวเช่นนี้ไม่ใช่ว่าให้ท่านผู้อ่านทำใจ (เพราะ "บัก" เป็นเรื่องธรรมชาติ อะไรทำนองนั้น) แต่เนื่องจากสาเหตุของการเกิดความผิดพลาดเกิดได้จากสาเหตุหลายประการ เช่น เกิดจากลอจิกภายในโปรแกรม หรือเกิดจากปัญหาของสภาพแวดล้อมดังที่กล่าวข้างต้น ไม่ว่าจะเป็นสาเหตุใดก็ตาม ผมคิดว่าการออกแบบโปรแกรมให้มีโครงสร้างที่ดีจะช่วยลดปัญหาการเกิดความผิดพลาด และง่ายต่อการทำความเข้าใจโปรแกรมเป็นอย่างมาก

ผมมีข้อแนะนำเล็กน้อยที่คิดว่าสามารถช่วยในการลดปัญหา และเพิ่มความเร็วในการตรวจสอบปัญหาดังนี้ครับ

- เขียนโปรแกรมให้อยู่ในรูปแบบโครงสร้างที่สื่อความหมายให้มากที่สุด

- ใช้การย่อหน้า หรือ TAB เพื่อกำหนดช่วงการทำงาน หรือสโคป (Scope) ของงานหรือโปรแกรมย่อย ดังตารางที่ 1

ตารางที่ 1 เปรียบเทียบการเขียนโปรแกรมแบบไม่มีโครงสร้าง และแบบมีโครงสร้าง

การเขียนโปรแกรมแบบไม่ใช้ย่อหน้ากำหนดช่วงการทำงานของโปรแกรม ทำให้โปรแกรมอ่านค่อนข้างยาก	การเขียนโปรแกรมโดยใช้ย่อหน้าช่วยกำหนดช่วงการทำงานของวนรอบ (loop) ในโปรแกรม ทำให้ง่ายต่อการเข้าใจโปรแกรม

`Procedure DoSmeThng(ptr:pointer);`
`Var wid, hi : integer;`
`Begin`
`With ptr^ do`
`Begin`
`For wid = 1 to width do`
`Begin`
`For hi = 1 to height*wid do`
`Begin`
`…`
`End;`
`…`
`End;`
`End;`
`End;`
	`Procedure DoSmeThng(ptr:pointer);`
	`Var wid, hi : integer;`
	`Begin`
	`With ptr^ do`
	`Begin`
	`For wid = 1 to width do`
	`Begin //Cal. Width`
	`For hi = 1 to height do`
	`Begin //Cal. Height`
	`…`
	`End;`
	`…`
	`End;`
	`End;`
	`End;`

● การกำหนดมาตรฐานงาน (ของตัวเรา) เช่น

- กำหนดชื่อของแบบข้อมูลอาจเริ่มต้นด้วยตัวอักษร T เช่น Tint = integer;

- กำหนดรูปแบบของตัวแปรชนิดอะเรย์ (Arrays) ด้วยตัวอักษร "S" ต่อท้าย เช่น TintS = array[0..100] of integer;

- กำหนดรูปแบบของตัวแปรชนิดค่าคงที่ (Const) ด้วยตัวอักษร "K" นำหน้า เช่น KDefInt = 0;

● ตั้งชื่อตัวแปร (Variables) ให้สื่อความหมายหรือใกล้เคียงกับสิ่งหรืองานที่ใช้มากที่สุด เช่น myComputerName = 'PP166MX'; อ่านแล้วสื่อความหมายมากกว่า s = 'PP166MX' ; หรืออาจจะใช้ในลักษณะตัวย่อ คือ myComNme = 'PP166MX';

● ใส่คำอธิบาย (Comment) ในบรรทัด หรือส่วนของโปรแกรมที่ต้องการคำอธิบายเพิ่มเติม

● ออกแบบโปรแกรมให้เป็นโปรแกรมย่อย ถ้ามีโปรแกรมย่อยมากให้จัดหมวดหมู่และแยกรวมโปรแกรมย่อยไว้เป็นยูนิต (Units) ต่างหาก ในกรณีการจัดหมวดหมู่โปรแกรมย่อยเป็นยูนิตต่างหาก นอกจากช่วยให้ตรวจสอบโปรแกรมได้ง่ายแล้ว ยังสามารถนำไปใช้ในโปรแกรมอื่น ๆ ได้อีก โดยไม่ต้องเขียนโค้ดซ้ำ

DEBUG

การค้นหาความผิดพลาดในระบบ หรือการ "ดีบัก" บน Delphi สามารถทำได้หลายวิธี เช่น ใช้วิธีการแสดงผลข้อมูล (Dump Data) ออก Output Device, ใช้ Complier Directives ช่วยตรวจสอบ, ใช้ Assertions Debugging Tools และใช้ Delphi's Integrated Debugger

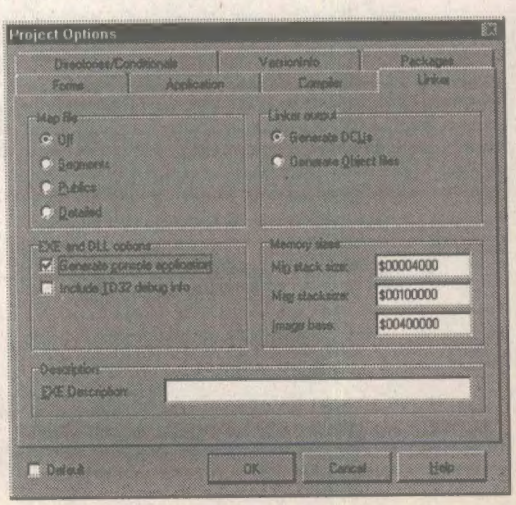
การแสดงผลข้อมูล (Dump Data) ออก Output Device

เป็นวิธีการที่สะดวก และง่ายที่สุด สามารถใช้กำหนดการแสดงผลได้ตามต้องการ โดยสามารถกำหนดให้แสดงผลได้หลายวิธีเช่น

● แสดงผลแบบ Message Box ด้วยคำสั่ง

```
procedure ShowMessage(const Msg: string);
```

● แสดงผลโดย Output ออก Console ด้วยคำสั่ง Write และ Writeln ซึ่งเป็นคำสั่งที่คุ้นกันดีสำหรับผู้ที่เคยพัฒนาโปรแกรมด้วยเทอร์โบปาสคาล การใช้งานบน Delphi จำเป็นที่จะต้องกำหนดให้ Linker สร้างหน้าต่าง Console เพื่อใช้ในการแสดงผล สามารถทำได้โดย เลือกเลือกเมนู Projects → Options → และเลือก Linker Page แล้วจึงกาเครื่องหมาย ☑ หน้าหัวข้อ Generate console application ดังรูปที่ 1



รูปที่ 1 การ Generate console application

```
procedure Dump2LogFile( content : shortString );
var fp : textFile; fileName : string;
begin
  fileName:='DUMPDATA.LOG';
  if ( fileExists( fileName ) ) then
  begin
    assignFile( fp,fileName ); append( fp );
  end else
  begin
    assignFile( fp,fileName ); rewrite( fp );
  end;
  writeln( fp, DateToStr(date),' : ',TimeToStr(time),' :: ',content );
  closeFIle( fp );
end;
```

รูปที่ 2
กิจกรรมบันทึกข้อมูลลง Text File

```
00001:  // Add this section to your project file (.DPR)
00002:  {$IFDEF DEBUG_MODE}
00003:  {$APPTYPE CONSOLE}
00004:  {$ENDIF}
00005:
00006:  // Sample procedure with conditional compilation directive
00007:  Procedure LoadData( fName : string; var dataP : TCustDataS );
00008:  var fp : file of TCustDataR; c : integer; data : TCustDataR;
00009:  begin
00010:    assignFile( fp, fName ); reset( fp ); c:=0;
00011:    while (not(eof(fp))) do
00012:    begin
00013:      readln( fp,data );
00014:  {$IFDEF DEBUG_MODE}
00015:      with data do
00016:        writeln( 'cnt=',c:5,' id= ',id:6,' name= ',name:30 );
00017:  {$ENDIF}
00018:      inc(c);
00019:  {$IFDEF DEBUG_MODE}
00020:      if (c<=MAX_CUST_DATA) then
00021:        move( data, dataP^[c], sizeof( TCustDataR ) );
00022:  {$ELSE}
00023:      move( data, dataP^[c], sizeof( TCustDataR ) );
00024:  {$ENDIF}
00025:    end;
00026:    closeFile(fp);
00027:  end;
```

รูปที่ 3
แสดงการใช้ Conditional Compilation Directive

● แสดงผลโดยเอาต์พุตออกเป็น Log file โดยบันทึกลงแฟ้มข้อมูลที่กำหนด ข้อดีของวิธีนี้คือสามารถบันทึกข้อมูลต่อเนื่องได้มาก และได้ทุกรายการ กิจกรรมที่ใช้แสดงอยู่ในรูปที่ 2

โดยปกติในการดีบักโปรแกรม เราจะเพิ่มส่วนโค้ดเมื่อต้องการดีบัก และลบโค้ดนั้นทิ้งเมื่อโปรแกรมสมบูรณ์ ในกรณีที่มีตำแหน่งที่ต้องการดีบักจำนวนมากก็อาจจะทำได้โดยไม่สะดวก หรือผิดพลาดได้ง่าย เราสามารถใช้ Compiler directive ชนิด Conditional compilation เพิ่มความสะดวกในการใช้ส่วนดีบักโปรแกรม ได้โดยเพิ่มโค้ด (code) ในโปรแกรมดังรูปที่ 3

อธิบายโปรแกรมในรูปที่ 3

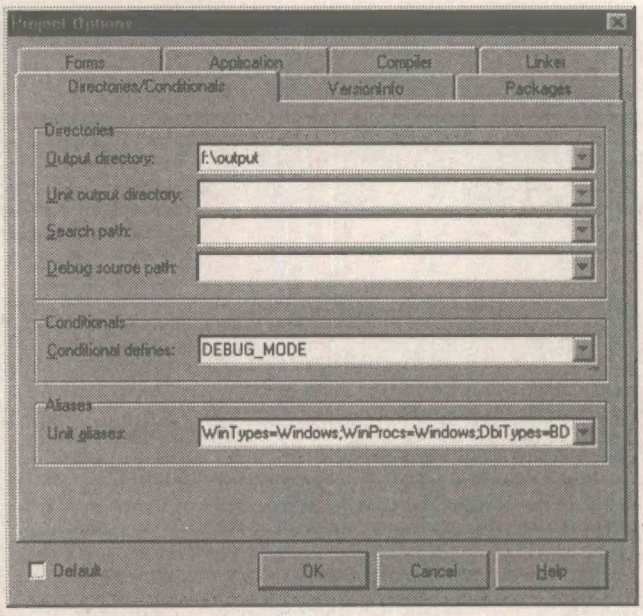
ในส่วนของ Project File จะต้องเพิ่มโค้ดตาม**บรรทัดที่ 2-4** ไว้ในตอนต้น เพื่อเป็นการระบุให้ Complier สร้างหน้าต่าง Application Console แทนการสร้างด้วยเมนูดังที่กล่าวแล้วข้างต้น

บรรทัดที่ 14-17 แสดงการใช้ Complier Directive ชนิด Conditional compilation แบบเงื่อนไขเดียว โดยถ้ามีการกำหนด symbol 'DEBUG_MODE' แล้วให้แสดงผลข้อมูลออก console

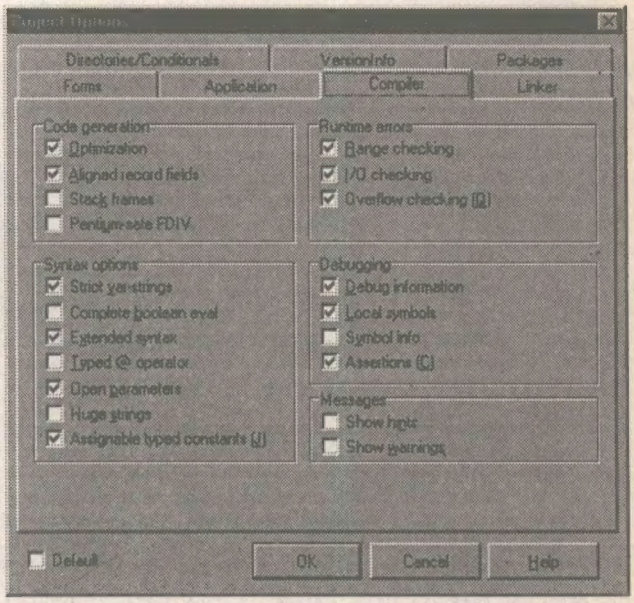
บรรทัดที่ 19-24 แสดงการใช้เงื่อนไขแบบมี ELSE ร่วม โดยถ้ามีการกำหนด symbol 'DEBUG_MODE' ให้ทำงานที่บรรทัดที่ 20-21 และถ้าไม่กำหนดให้ทำงานที่บรรทัดที่ 23

การกำหนดเงื่อนไขที่ระบุใน Conditional compilation ทำได้ 2 วิธีคือ

จากเมนู Delphi เลือกเมนู Projects → Options และในหน้าต่าง Project Options หน้า Directories/Conditional ให้ระบุเงื่อนที่ต้องการในช่อง Conditional defines ดังรูปที่ 4 หรือกำหนดโค้ดเงื่อนไขไว้ในส่วนต้นไฟล์ Project (.DPR) โดยตรง เช่น



รูปที่ 4 การกำหนด Conditional



รูปที่ 5 การใช้งาน Complier Directives และ Assertions

```
{$DEFINE DEBUG_MODE}
```

จะทำให้มีผลบังคับใช้ทั้งโปรเจ็กต์ ในกรณีต้องการยกเลิกการกำหนด symbol 'DEBUG_MODE' ภายในโปรแกรม ให้ใช้คำสั่ง {$UNDEF DEBUG_MODE} ณ ตำแหน่งที่ต้องการ

การใช้ Complier Directives และ Assertions Debugging Tools

ผมขออธิบายการใช้ Complier Directives และ Assertions Debugging Tools ไว้ในเรื่องเดียวกัน เนื่องจากในส่วนของ Assertions จำเป็นต้องใช้ Complier Directives ประกอบด้วยครับ

Complier Directives มีจำนวนมากมายหลายชนิด ผมขออนุญาตกล่าวถึงเฉพาะส่วนที่สามารถนำมาใช้ในงาน Debug ก่อน สำหรับส่วนอื่นถ้ามีโอกาสจะนำมาเสนอในคราวหน้าครับ Complier Directives ที่สามารถนำมาใช้ในงาน Debug จะมีอยู่ 3 ชนิด (ไม่รวม Assertions) คือ Range cheking {$R}, I/O checking {$I} และ Overflow checking {$Q}

Range checking {$R} เป็น Complier Directives สำหรับตรวจสอบการอ้างถึงข้อมูลในอะเรย์ ว่ามีการอ้างถึงอะเรย์เกินกว่าที่มีอยู่จริงหรือไม่ ถ้าเกิดความผิดพลาดขึ้นโปรแกรมจะแจ้ง Range check error (หรือ ERangeError Exception) ขึ้น ณ ตำแหน่งที่ผิดพลาด (ในกรณีที่รันโปรแกรมภายใต้ IDE)

เราสามารถ enable ส่วนการตรวจสอบได้โดยใส่เครื่องหมาย ☑ ในหน้าต่าง Project Options ในส่วนของ Runtime errors ในช่อง Range checking ดังรูปที่ 5 หรือเพิ่ม Complier Directive symbol {$R+} ไว้ในส่วนต้นของโปรเจ็กต์ไฟล์ (.DPR) และเมื่อต้องการยกเลิก (disable) ให้ใช้ {$R-} การใช้ Complier Directives ชนิดนี้ก่อให้เกิดผลข้างเคียง คือ จะทำให้โปรแกรมมีขนาดใหญ่ขึ้น และทำงานช้าลง (ดังนั้นเมื่อทำการ Debug เรียบร้อยควรจะ disable ส่วนนี้)

I/O checking {$I} เป็น Complier Directives สำหรับตรวจสอบการใช้งานส่วน I/O ของระบบ โดยเมื่อ enable จะแจ้งข้อความผิดพลาด I/O Error (หรือ EInOutError Exception) พร้อมหมายเลข error ในกรณีที่เกิดความล้มเหลวในการทำงานของ I/O เช่นการเปิด/ปิดไฟล์ ถ้า disable จะต้องใช้คำสั่ง Ioresult ตรวจสอบหมายเลข error เอง

การ enable ในส่วนของ Project Options สามารถทำได้เหมือนกับ Range checking ในช่อง I/O checking หรือใช้ Complier Directive symbol {$I+} และใช้ {$I-} เมื่อต้องการ disable

Overflow checking {$Q} เป็น Complier Directives สำหรับตรวจสอบการเกิด Overflow เมื่อใช้คำสั่งการคำนวณต่าง ๆ (arithmetic operations) โดยเมื่อเกิดความผิดพลาดจะแจ้งข้อความ error (หรือ EIntOverflow Exception) การ enable ทำได้เช่นเดียวกับ Range checking ในช่อง Overflow checking

ผลข้างเคียงการใช้ Complier Directive ชนิดนี้ จะทำให้โปรแกรมใหญ่ขึ้น และการทำงานช้าลงเหมือนกับ Range checking directive

Assertions {$C} {$ASSERTIONS} Complier Directives ตัวใหม่ที่มีใน Delphi 3 ใช้สำหรับตรวจสอบเงื่อนไขชนิด Boolean ควบคู่กับคำสั่ง Assert(<boolean expression>); เช่น

```
Procedure TForm1.CopyData(dataP:pChar; dataBlk:TdataBlk);
var len : integer;
begin
    len:=strlen( dataP );
    Assert( len<sizeof( TdataBlk ) );
    move( dataP^, dataBlk^, len );
end;
```

โดยถ้าเงื่อนไขเป็นเท็จ (FALSE) จะแสดง Message Box แจ้งพบข้อความผิดพลาดพร้อมหมายเลขบรรทัดที่ผิด และยุติการทำงานทันทีดังรูปที่ 6

การใช้ Assertion สามารถกำหนดได้ในหน้าต่าง Project Options ในส่วน Debugging ดังรูปที่ 5 หรือใช้ Complier Directives ชนิด {$C+} หรือ {$ASSERTIONS ON} ไว้ในส่วนต้นของโปรเจ็กต์ไฟล์ (.DPR)

ประโยชน์ที่เห็นเด่นชัดของ Assertion Directive คือการใช้ Assertion Switch ({$C} หรือ {ASSERTIONS}) ควบคุมการทำงานของคำสั่ง Assert กล่าวคือ เมื่อได้ทำการยกเลิกการใช้ Assertion Directive ด้วย {$C-} หรือ {$ASSERTIONS OFF} แล้ว Complier จะยกเลิกการแปลคำสั่ง Assert ที่มีอยู่ในโปรแกรมทั้งหมด โดยเราไม่จำเป็นต้องลบบรรทัดคำสั่ง Assert ออกทั้งหมด ทำให้การพัฒนาโปรแกรมเป็นไปได้อย่างรวดเร็วยิ่งขึ้น

การใช้ Delphi's Integrated Debugger

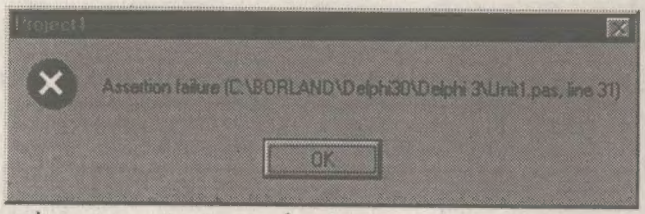
Delphi เป็นคอมไพเลอร์ที่รวมเอาระบบ Debugger เข้าไว้ในส่วนการพัฒนาโปรแกรม หรือ IDE (Integrated Development Environment) ทำให้มีความคล่องตัวสูงในการตรวจสอบหาข้อผิดพลาดในโปรแกรม สามารถตรวจสอบโปรแกรม (Tracing) ในลักษณะ "On the fly" ได้ละเอียดถึงระดับคำสั่งต่อคำสั่ง

เปิดระบบ Integrated Debugger

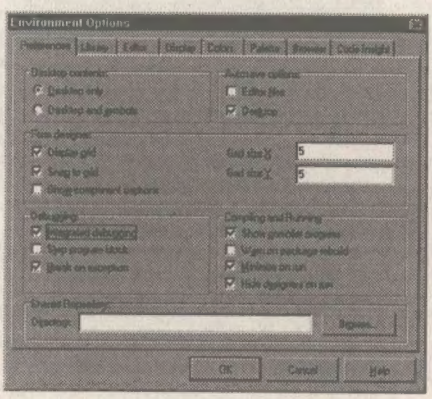
การใช้งานระบบ Delphi's Integrated Debugger จำเป็นต้องกำหนดให้ IDE เปิดการใช้งานระบบนี้ สามารถทำได้โดยกำหนดค่าตัวเลือกในเมนู Tools → Environment Options → หน้า Perferences ในส่วน Debugging ตามรูปที่ 7 ดังนี้

Integrated Debugging กาเครื่องหมาย ☑ ไว้หน้าหัวข้อนี้ เพื่อกำหนดให้ IDE ใช้ระบบ Integrated Debugging

Break on exception กาเครื่องหมาย ☑ ถ้าพบความผิดพลาดให้โปรแกรมหยุดชั่วขณะ (Pause) และแสดงหน้าต่าง Error Exception



รูปที่ 6 แสดงข้อความผิดพลาดเมื่อ Assertion fail



รูปที่ 7 การกำหนดให้ IDE ใช้ระบบ Integrated Debugging

กำหนดเงื่อนไขการคอมไพล์โปรแกรม

นอกจากการกำหนดให้ IDE ใช้ feature ดังกล่าวแล้ว จะต้องกำหนดให้คอมไพเลอร์เพิ่มข้อมูลส่วนดีบักลงในโปรแกรม สามารถทำได้โดยเลือกเมนู Project → Options → หน้า Complier ในกรอบ Debugging ให้กาเครื่องหมาย ☑ ในหัวข้อ Debug Information และ Local Symbols ดังรูปที่ 5

การใช้งาน และโปรแกรมทดสอบ

การใช้งานระบบ Integrated Debugger มีเทคนิคและวิธีการใช้ค่อนข้างละเอียดมาก ผมจะกล่าวถึงเฉพาะบางส่วนที่จำเป็นเท่านั้นครับ โดยปกติการใช้งานสามารถทำได้โดยผ่านเมนู Run หรือการคลิกขวาที่ซอร์สโค้ดของโปรแกรม ผมได้เตรียมโปรแกรมตัวอย่างสั้น ๆ สำหรับศึกษาลำดับการทำงานของระบบ Integrated Debugger ควบคู่กันไป ตัวโปรแกรมแสดงอยู่ในลิสติงที่ 1 เราสามารถสั่งให้โปรแกรมรันในรูปแบบต่าง ๆ ดังต่อไปนี้ครับ

Run หรือ [F9] คำสั่งให้โปรแกรมรันตามปกติ เป็นคำสั่งที่เราทราบกันดีอยู่แล้ว แต่สามารถใช้เป็นคำสั่งให้โปรแกรมทำงานต่อ (continue) หลังจากใช้คำสั่ง Program Pause

Program Pause คำสั่งให้โปรแกรมหยุดทำงานชั่วขณะ

ลิสติงที่ 1 โปรแกรมทดสอบการใช้ IDE : Integrated Debugger

```
00001:  // Listing 1.1) IDE : Integrated Debugger Demo
00002:  // uIDEdemo.pas
00003:  unit uIDEdemo;
00004:  interface
00005:  uses
00006:    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
00007:    StdCtrls, Buttons;
00008:  type
00009:    TFideDemo = class(TForm)
00010:      BitBtn1: TBitBtn;
00011:      Memo1: TMemo;
00012:      procedure FormCreate(Sender: TObject);
00013:      procedure FormCloseQuery(Sender: TObject; var CanClose:Boolean);
00014:      procedure BitBtn1Click(Sender: TObject);
00015:    private
00016:      { Private declarations }
00017:        cnt : integer;
00018:      procedure InitVar;
00019:      procedure ReleaseVar;
00020:      procedure Echo( content : string );
00021:    public
00022:      { Public declarations }
00023:    end;
00024:  var
00025:    FideDemo: TFideDemo;
00026:  implementation
00027:  {$R *.DFM}
00028:  //- Initialized section ---------------------
00029:  procedure TFideDemo.InitVar;
00030:  begin
00031:    cnt:=0;
00032:  end;
00033:  procedure TFideDemo.ReleaseVar;
00034:  begin
00035:  // nothing do
00036:  end;
00037:  //-Addition method ------------------------
00038:  procedure TFideDemo.Echo( content : string );
00039:  begin
00040:    if ( memo1.lines.count>300 ) then memo1.lines.delete(0);
00041:    memo1.lines.add(content);
00042:  end;
00043:  //---------------------------------
00044:  procedure TFideDemo.FormCreate(Sender: TObject);
00045:  begin
00046:    InitVar;
00047:  end;
00048:  procedure TFideDemo.FormCloseQuery(Sender: TObject;
00049:                                     var CanClose: Boolean);
00050:  begin
00051:    ReleaseVar; CanClose:=true;
00052:  end;
00053:  procedure TFideDemo.BitBtn1Click(Sender: TObject);
00054:  var c : integer;
00055:  begin
00056:    for c:= 1 to 10 do
00057:    begin
00058:      inc( cnt );
00059:      Echo( 'cnt = '+IntTostr( cnt ) );
00060:    end;
00061:  end;
00062:  end.
```

Program Reset หรือ [Ctrl] + [F2] คำสั่งให้โปรแกรมยุติการทำงานโดยถาวร เป็นคำสั่งที่มีประโยชน์มาก ผมมักใช้เวลาผมติดอยู่ใน Infinited loop

Step Over หรือ [F8] คำสั่งให้โปรแกรมทำงานทีละคำสั่ง โดยไม่มีการ Trace เข้าในโปรแกรมย่อยหรือฟังก์ชันย่อยที่อยู่ภายในกิจกรรม เราสามารถทดลองใช้กับโปรแกรมทดสอบได้ดังนี้

● เตรียมโปรแกรมเรียบร้อย

● กดปุ่ม [F8] เพื่อสั่งให้ระบบดีบักเกอร์ทำงานในลักษณะ Step Over โปรแกรมจะเริ่มรันโดยแสดง Highlight ที่ตำแหน่งบรรทัดสุดท้าย

● กดปุ่ม [F8] อีกครั้ง โปรแกรมจะเริ่มทำงานในลักษณะลำดับขั้น (Tracing) แบบ Step Over กล่าวคือ *ระบบดีบักเกอร์จะทำงานทุกกิจกรรมที่อยู่ภายใต้ Events ต่าง ๆ ภายในโปรแกรม โดยไม่ลำดับการทำงานกลไกภายในกิจกรรมนั้น* สำหรับโปรแกรมทดสอบจะเริ่มการทำงานที่กิจกรรม InitVar ใน FormCreate Events

● กดปุ่ม [F8] ระบบดีบักเกอร์จะ Trace ไปที่บรรทัด 'End' ของ FormCreate เป็นการจบ FormCreate Events

● เนื่องจาก Application ที่ทำงานอยู่ภายใต้ระบบวินโดวส์มีลักษณะเป็น Events Driven ดังนั้นเราจำเป็นต้องตอบสนองกับโปรแกรมทดสอบเพื่อให้เกิด Events โดยคลิกที่ปุ่ม 'Count' ระบบดีบักเกอร์จะเริ่มทำงานต่อทันที

● กดปุ่ม [F8] เพื่อศึกษาลำดับการทำงานต่อไป

Trace in to หรือ [F7] คำสั่งให้โปรแกรม Trace การทำงานทุกโปรแกรมย่อย หรือฟังก์ชันย่อยในกิจกรรมที่มีข้อมูล Debug Information ดังนี้

● กดปุ่ม [F7] เพื่อให้ระบบดีบักเกอร์ทำงานในลักษณะ Trace in to

● โปรแกรมจะเริ่มทำงานเหมือนกับใช้ระบบดีบักแบบ Step Over

● เมื่อโปรแกรมทำงานถึงกิจกรรม InitVar และกดปุ่ม [F7] แล้วระบบดีบักเกอร์จะเริ่มทำงานตามลำดับคำสั่ง หรือชุดโปรแกรมย่อย ของกิจกรรม InitVar กล่าวคือ *ระบบดีบักจะทำงานทุกกิจกรรมที่อยู่ภายใต้ Events และยังคงลำดับการทำงานของกลไกต่าง ๆ ในกิจกรรม ถ้ากิจกรรมนั้น ๆ มี Debug Information* (คำสั่งมาตรฐานที่ Delphi เตรียมไว้ให้จะไม่มี Debug Information ระบบดีบักเกอร์จึงไม่สามารถลำดับการทำงานได้)

● กดปุ่ม [F7] เพื่อศึกษาลำดับการทำงานต่อไป

Run to cursor หรือ [F4] คำสั่งกำหนดให้โปรแกรมทำงานจนถึงบรรทัดที่เลือกไว้ การกำหนดตำแหน่ง cursor สามารถทำได้โดยการคลิกซ้ายที่บรรทัดใด ๆ

Show execution Point ให้คอมไพเลอร์แสดงซอร์สโค้ด ณ ตำแหน่งปัจจุบันที่โปรแกรมรันอยู่

```
procedure TFideDemo.Echo( content : string );
begin
  if ( memo1.lines.count > 30 ) then memo1.lines.delete(0);
  memo1.lines.add(content);
end;                    content = 'cnt = 1'
//-----------------------------------------------------------
```

รูปที่ 8 แสดงการดูค่าตัวแปรแบบ "On the fly"

Add Break Point หรือ Toggle Break Point สั่งให้โปรแกรมหยุด (Pause) ณ ตำแหน่งบรรทัดที่กำหนด สามารถทำได้โดยกดปุ่ม [Ctrl]+[F8]

ถ้าต้องการยกเลิกการกำหนด Break Point ให้กดปุ่ม [Ctrl]+[F8] อีกครั้ง

ในขณะที่รันโปรแกรมด้วยระบบ Integrated Debugger เราสามารถใช้เมาส์คลิกที่ตำแหน่งตัวแปรใด ๆ ในโปรแกรม เพื่อขอทราบค่าที่บรรจุอยู่ในตัวแปรได้ทันทีดังรูปที่ 8 (ตัวแปรนั้นต้องผ่านขั้นตอนการทำงานเรียบร้อยแล้ว)

Exception Handling

การเกิดความผิดพลาด (Exception) ในระบบโปรแกรม หรือที่รู้จักกันดีในนาม "Run-time error" บนระบบ DOS เป็นสิ่งที่นักพัฒนาระบบทุกคนไม่ปรารถนา การเกิด Exception สำหรับนักพัฒนาระบบอาจเป็นเพียงการสูญเสียเวลาในการค้นหาข้อผิดพลาดในโปรแกรม แต่สำหรับองค์กรแล้วนั้นหมายถึงการสูญเสียในเชิงธุรกิจ ไม่ว่าจะเป็นสูญเสียเวลา ข้อมูล และค่าใช้จ่ายในด้านต่าง ๆ ซึ่งอาจจะกระทบต่อเสถียรภาพความน่าเชื่อถือขององค์กร

การแก้ไขปัญหาที่คาดว่าจะเกิดขึ้นของ Exception อาจทำได้ไม่ทั้งหมด เนื่องจากปัญหาอาจจะเกิดขึ้นได้จากหลาย ๆ สาเหตุ เช่น

● จาก Components ต่าง ๆ ที่ใช้ในระบบ

● ดาต้าเบสไดรเวอร์ (Database Driver) เช่น BDE หรือ ODBC

● ดีไวซ์ไดรเวอร์ (Device Drivers) ต่าง ๆ ที่อยู่ในระบบ เช่น VGA Card Driver, Sound Card Driver

● ไฟล์ระบบ Networks, ไฟล์ Dynamic Link Library (DLL), โปรแกรมตรวจสอบไวรัส หรือโปรแกรมยูทิลิตี้ต่าง ๆ

● Delphi Run-Time Library (RTL) หรือ Visual Component Library (VCL)

● และ ตัวโปรแกรมระบบปฏิบัติการ (วินโดวส์) เอง

ระบบวินโดวส์ 95 และวินโดวส์ NT ได้เตรียมส่วน Exception Architecture ไว้ ทำให้สามารถตรวจสอบและรายงานข้อผิดพลาดได้ โดยสามารถสร้างกิจกรรมขึ้นรองรับหรือสับเปลี่ยนขั้นตอนการทำงานของโปรแกรมเมื่อเกิดข้อผิดพลาดไปยังกิจกรรมที่เตรียมไว้ เพื่อแจ้งข้อความเตือน หรือดำเนินการแก้ไข

ทำไมต้องใช้ Exception Handler

การเขียนโปรแกรมเพื่อตอบสนอง Exception Architecture มีจุดประสงค์หลักคือต้องการให้โปรแกรมมีเสถียรภาพมากที่สุด (Robust Applications) ซึ่งมีวิธีการดำเนินการอยู่ 2 แบบคือ

1. Executing cleanup code เพื่อทำการเคลียร์ตัวแปรต่าง ๆ ให้อยู่ในสภาวะที่เหมาะสม โปรแกรมโดยทั่วไปมักจะมีการใช้หน่วยความจำหรือทรัพยากรของระบบ (System Resources) การเคลียร์ค่าต่าง ๆ หมายถึงการยกเลิกการใช้หน่วยความจำและทรัพยากรเพื่อคืนให้กับระบบ ทำให้โปรแกรมอื่น ๆ ที่รันอยู่บนระบบปฏิบัติการเดียวกันสามารถทำงานต่อไปได้โดยไม่มีปัญหา

2. Handling the exception เพื่อทำการแก้ไขหรือยกเลิกส่วนที่ผิดพลาด ทำให้โปรแกรมสามารถดำเนินการต่อไปได้ การเกิดความผิดพลาดบางชนิดอาจจะสามารถแก้ไขหรือยกเลิกความผิดพลาด เช่น การไม่สามารถเปิดแฟ้มข้อมูลได้เนื่องจากไม่มีแฟ้มข้อมูลนั้นในระบบ การเปลี่ยนไดเรคทอรีปัจจุบันไปยังไดเรคทอรีที่ไม่มีในระบบ ในส่วนนี้เราสามารถแก้ไขได้โดยยกเลิกการเกิดข้อความผิดพลาดและแสดงข้อความเตือนที่หน้าจอภาพ

โปรแกรมอย่างไร

การเขียนโปรแกรมในลักษณะ Exception Handling จะต้องสร้างส่วนที่เรียกว่า "Protection Block" คือส่วนที่ใช้ปกป้องโปรแกรมในส่วนที่คิดว่าสามารถเกิดความผิดพลาดได้ง่าย

การพิจารณาว่าโปรแกรมในส่วนใดควรจะป้องกันการเกิดความผิดพลาดไม่มีหลักการตายตัว แต่สามารถทำได้โดยพิจารณาว่าโปรแกรมส่วนไหนต้องมีการติดต่อกับ I/O (Input/Output) มีการจองหน่วยความจำ (Allocated Memory) ใช้รีซอร์สของวินโดวส์ (Windows Resource) หรือใช้ออบเจ็กต์อื่น (Objects) บ้าง ซึ่งเป็นเหตุให้เกิดความผิดพลาดได้ทั้งสิ้น ตัวอย่างเช่น

กิจกรรม procedure TFexcpHdl.NoException; เป็นกิจกรรมที่เขียนขึ้นโดยไม่ใช้ Exception Handling เมื่อกิจกรรมดำเนินการและทำการเปิดไฟล์ชื่อ "Noname.txt" ถ้าไม่พบไฟล์ที่ในไดเรคทอรีปัจจุบันก็จะเกิดข้อผิดพลาด (Run-time Error) และโปรแกรมจะยุติการทำงาน ณ ตำแหน่งที่เกิดข้อผิดพลาดทันที (คำสั่ง Reset(fp)) ทำให้การทำงานของโปรแกรมเป็นไปอย่างไม่สมบูรณ์ เนื่องจากไม่สามารถยกเลิกหน่วยความจำที่จองให้กับพอยน์เตอร์ p ได้ ระบบปฏิบัติการจึงสูญเสียเนื้อที่หน่วยจำโดยไม่จำเป็น

```
procedure TFexcpHdl.NoException;
var p : pointer; fp : file;
begin
  Echo( 'NoException routine.' ); Echo( #32 );
  Echo( 'Get memory' );
  getmem( p, sizeof( shortString ) );
  // จองหน่วยความจำ
  Echo( 'Do something cause an error..' );
  assignFile( fp,'noname.txt' ); reset(fp);   closeFile(fp);
  // เปิดแฟ้มข้อมูลที่อาจจะไม่มีอยู่ในระบบ
  Echo( 'Release memory' );  freemem( p, sizeof
  ( shortString ) ); // ยกเลิกหน่วยความจำ
end;
```

Executing Cleanup Code

ตัวอย่างสำหรับการใช้ Exception Handling ชนิด Excuting Cleanup Code ควบคุมการเกิดข้อผิดพลาดด้วยคำสั่ง "try..finally" ซึ่งมีความหมายคือ ให้ปกป้องกิจกรรมใด ๆ ที่อยู่ระหว่าง try..finally และบังคับให้ดำเนินการตามกิจกรรมที่อยู่ในช่วง "finally..end" ไม่ว่าโปรแกรมจะเกิดข้อผิดพลาดหรือไม่ก็ตาม จากโปรแกรมเราจะเห็นว่ากิจกรรมได้ใส่คำสั่งเปิดแฟ้มข้อมูลไว้ระหว่าง "try..finally" และใส่คำสั่งยกเลิกหน่วยความจำไว้ในช่องของ "finally..end" เมื่อรันโปรแกรมและพบข้อผิดพลาด กิจกรรมจะทำการยกเลิกหน่วยความจำก่อน แล้วจึงแสดงข้อความผิดพลาด และยุติการทำงาน

```
procedure TFexcpHdl.CleanUpCode;
var p : pointer; fp : file;
begin
  Echo( 'Excuting with cleanup code.' ); Echo( #32 );
  Echo( 'Get memory' );
  getmem( p, sizeof( shortString ) );
  try
    assignFile( fp,'noname.txt' ); reset(fp); closeFile(fp);
  finally
    Echo( 'Release memory' );  freemem( p, sizeof( shortString ) );
  end;
end;
```

Handling the exception

ลองพิจารณากิจกรรมอีกส่วนที่ใช้ Exception Handling แบบควบคุมส่วนที่เกิดความผิดพลาด ด้วยวิธีนี้กิจกรรมจะใช้คำสั่ง "try..except" เพื่อควบคุมการเกิดข้อผิดพลาด กิจกรรมจะใส่ชุดคำสั่งเปิดแฟ้มข้อมูลไว้ระหว่าง "try..except" และใส่คำสั่งที่ต้องการให้ตอบสนองการเกิดข้อผิดพลาดไว้ระหว่าง "except..end" ซึ่งกิจกรรมจะทำงานในส่วนนี้ก็ต่อเมื่อเกิดความผิดพลาดเท่านั้น ในที่นี้ใช้การตรวจสอบ I/O Error ด้วย EinOutError Class และแสดงเมสเสจบอกถ้าเกิดข้อผิดพลาดครั้นเมื่อโปรแกรมรันและเกิดข้อผิดพลาด โปรแกรมจะแสดง Message ข้อผิดพลาดของ EinOutError Object แล้วจึงทำงานต่อจากช่วงคำสั่ง "except..end" ทำให้โปรแกรมสามารถรันต่อไปได้โดยไม่ถูกขัดจังหวะจากการเกิดข้อความผิดพลาด

```
procedure TFexcpHdl.HandleException;
var p : pointer; fp : file;
begin
  Echo( 'Handle the exception.' ); Echo( #32 );  Echo( 'Get memory' );
  getmem( p, sizeof( shortString ) );
  try
    assignFile( fp,'noname.txt' ); reset(fp); closeFile(fp);
  except
    on E: EInOutError do echo( E.Message  );
  end;
  Echo( 'Release memory' );
  freemem( p, sizeof( shortString ) );
end;
```

Exception Class

Exception Class เป็น Objects ต้นแบบที่สืบสกุลจาก Tobjects สำหรับ Exception Class ทั้งหมดที่มีอยู่ใน Delphi 3.0 (บรรจุอยู่ในยูนิต SysUtils.pas) แสดงอยู่ในตารางที่ 2 สร้างขึ้นเพื่อตอบสนองการใช้งาน Exception Handling Type ต่าง ๆ โดยผู้พัฒนาระบบสามารถสร้าง Except Handler Class สืบสกุลใหม่ได้โดยตรงจาก Exception Objects สำหรับกิจกรรมที่สำคัญมีดังต่อไปนี้

```
type
 Exception = class(TObject)
 public
   constructor Create(const Msg: string);
   constructor CreateFmt(const Msg: string;
   const Args: array of const);
   constructor CreateRes(Ident: Integer);
   constructor CreateResFmt(Ident: Integer;
   const Args: array of const);
   constructor CreateHelp(const Msg: string; HelpContext: Integer);
   constructor CreateFmtHelp(const Msg: string;
   const Args: array of const; HelpContext: Integer);
   constructor CreateResHelp(Ident, HelpContext: Integer);
   constructor CreateResFmtHelp(Ident: Integer;
   const Args: array of const;  HelpContext: Integer);
   property HelpContext: Integer;
   property Message: string;
 end;
```

Exception Objects มี property 2 ตัวคือ HelpContext สำหรับอ้างถึง Help Index และ Message สำหรับบรรจุข้อความ และมี constructor หลายรูปแบบสำหรับการ Initialized ตอบสนองในเงื่อนไขต่าง ๆ กัน โดย

Constructor ที่มีคำ "Fmt" ต่อท้าย ค่า "Msg" ที่ผ่านให้ต้องอยู่ในรูปแบบ Format String มี "Args" ผ่านค่าให้

Constructor ที่มีคำ "Res" ต่อท้าย ค่า "Msg" จะบรรจุอยู่ใน Resource File ที่ลำดับ ident

และ Constructor ที่มีคำ "Help"ต่อท้าย จะต้องมีค่า HelpContext ประกอบกับ "Msg" ที่ผ่านให้

เราสามารถสร้าง Exception Class ใหม่ได้ด้วยการสืบสกุลจาก Exception Class ดังนี้

```
type
   EmyException = class( Exception );
```

และเรียกใช้งาน (raise) โดยในส่วนของ Protection block ดังต่อไปนี้

```
try
 ...
except
 on E : EmyException do begin ... end;
end;
```

ตารางที่ 2 แสดง Exception Class ทั้งหมดที่มีอยู่ในยูนิต SysUtils.Pas

Exception Class ในระบบ	
EAbort = class(Exception);	EAccessViolation = class(Exception);
EOutOfMemory = class(Exception)	EPrivilege = class(Exception);
EInOutError = class(Exception)	EStackOverflow = class(Exception);
EIntError = class(Exception);	EControlC = class(Exception);
EDivByZero = class(EintError);	EVariantError = class(Exception);
ERangeError = class(EintError);	EPropReadOnly = class(Exception);
EIntOverflow = class(EintError);	EPropWriteOnly = class(Exception);
EMathError = class(Exception);	EExternalException = class(Exception)
EInvalidOp = class(EmathError);	EAssertionFailed = class(Exception);
EZeroDivide = class(EmathError);	EAbstractError = class(Exception);
EOverflow = class(EmathError);	EIntfCastError = class(Exception);
EUnderflow = class(EmathError);	EInvalidContainer = class(Exception);
EInvalidPointer = class(Exception);	EInvalidInsert = class(Exception);
EInvalidCast = class(Exception);	EPackageError = class(Exception);
EConvertError = class(Exception);	EWin32Error = class(Exception);

การใช้งานและโปรแกรมทดสอบ

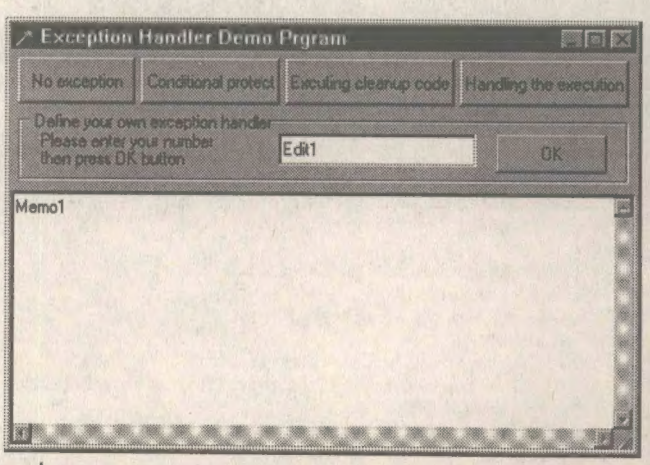
ผมสร้างโปรแกรมทดสอบและเปรียบเทียบการใช้ Exception Handling ในแบบต่าง ๆ ดังรูปที่ 9 และตัวโปรแกรมแสดงอยู่ในลิสติงที่ 2

วิธีการใช้โปรแกรม

No exception : แสดงการทำงานของกิจกรรมที่ไม่ใช้ Exception Routine

Conditional protect: แสดงการทำงานแบบตรวจสอบเงื่อนไขแทนการใช้ Exception Handling

Excuting cleanup code : แสดงการทำงานแบบ Cleanup code



รูปที่ 9 โปรแกรมทดสอบ Exception Handling

ลิสติงที่ 2 โปรแกรมทดสอบการใช้ Exception Handling

```
00001:  // Exception Handler Demo Program
00002:  // UEXCPHDL.PAS
00003:  unit uExcpHdl;
00004:  interface
00005:  {$H-$I+$R+$Q+}
00006:  uses
00007:    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
00008:    StdCtrls, ExtCtrls;
00009:  type
00010:    EWrongNumber = class(Exception);
00011:    TFexcpHdl = class(TForm)
00012:      Button1: TButton;
00013:      Button2: TButton;
00014:      Button3: TButton;
00015:      Memo1: TMemo;
00016:      GroupBox1: TGroupBox;
00017:      Edit1: TEdit;
00018:      Button4: TButton;
00019:      Label2: TLabel;
00020:      Label1: TLabel;
00021:      Button5: TButton;
00022:      procedure FormCreate(Sender: TObject);
00023:      procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
00024:      procedure Button1Click(Sender: TObject);
00025:      procedure Button5Click(Sender: TObject);
00026:      procedure Button2Click(Sender: TObject);
00027:      procedure Button3Click(Sender: TObject);
00028:      procedure Button4Click(Sender: TObject);
00029:    private
00030:      { Private declarations }
00031:      procedure InitVar;
00032:      procedure ReleaseVar;
00033:      procedure Schedule;
00034:      procedure Echo( content : shortString );
00035:      // exception method
00036:      procedure NoException;
00037:      procedure ConditionProtect;
00038:      procedure CleanUpCode;
00039:      procedure HandleException;
00040:      procedure RaiseYourException;
00041:    public
00042:      { Public declarations }
00043:    end;
00044:  var
00045:    FexcpHdl: TFexcpHdl;
00046:  implementation
00047:  {$R *.DFM}
00048:  //-------------------------------------
00049:  procedure TFexcpHdl.InitVar;
00050:  begin
00051:  // not used
00052:  end;
00053:  procedure TFexcpHdl.ReleaseVar;
00054:  begin
00055:  // not used
00056:  end;
00057:  procedure TFexcpHdl.Schedule;
00058:  begin
00059:    Application.ProcessMessages;
00060:  end;
00061:  procedure TFexcpHdl.Echo( content : shortString );
00062:  begin
00063:    if ( memo1.lines.count>255 ) then memo1.lines.delete(0);
00064:    memo1.lines.add( content ); Schedule;
00065:  end;
00066:  //-------------------------------------
00067:  procedure TFexcpHdl.NoException;
00068:  var p : pointer; fp : file;
```

→

ลิสติงที่ 2 (ต่อ)

```
00069:  begin
00070:    Echo( 'NoException routine.' ); Echo( #32 );
00071:    Echo( 'Get memory' );
00072:    getmem( p, sizeof( shortString ) );
00073:    Echo( 'Do something cause an error..' );
00074:    assignFile( fp,'noname.txt' ); reset(fp);
00075:    closeFile(fp);
00076:    Echo( 'Release memory' );
00077:    freemem( p, sizeof( shortString ) );
00078:  end;
00079:  procedure TFexcpHdl.ConditionProtect;
00080:  var p : pointer; fp : file;
00081:  begin
00082:    Echo( 'Conditional protected.' ); Echo( #32 );
00083:    Echo( 'Get memory' );
00084:    getmem( p, sizeof( shortString ) );
00085:    Echo( 'Do something cause an error..' );
00086:    if ( fileExists( 'noname.txt' ) ) then
00087:    begin
00088:      assignFile( fp,'noname.txt' ); reset(fp);
00089:      closeFile(fp);
00090:    end;
00091:    Echo( 'Release memory' );
00092:    freemem( p, sizeof( shortString ) );
00093:  end;
00094:  procedure TFexcpHdl.CleanUpCode;
00095:  var p : pointer; fp : file;
00096:  begin
00097:    Echo( 'Excuting with cleanup code.' ); Echo( #32 );
00098:    Echo( 'Get memory' );
00099:    getmem( p, sizeof( shortString ) );
00100:    try
00101:      assignFile( fp,'noname.txt' ); reset(fp);
00102:      closeFile(fp);
00103:    finally
00104:      Echo( 'Release memory' );
00105:      freemem( p, sizeof( shortString ) );
00106:    end;
00107:  end;
00108:  procedure TFexcpHdl.HandleException;
00109:  var p : pointer; fp : file;
00110:  begin
00111:    Echo( 'Handle the exception.' ); Echo( #32 );
00112:    Echo( 'Get memory' );
00113:    getmem( p, sizeof( shortString ) );
00114:    try
00115:      assignFile( fp,'noname.txt' ); reset(fp);
00116:      closeFile(fp);
00117:    except
00118:      on E: EInOutError do echo( E.Message  );
00119:    end;
00120:    Echo( 'Release memory' );
00121:    freemem( p, sizeof( shortString ) );
00122:  end;
00123:  procedure TFexcpHdl.RaiseYourException;
00124:  var s : string;
00125:  begin
00126:    s:=Edit1.text;
00127:    if ( s<>'3456' ) then
00128:      raise EWrongNumber.Create( 'Invalid number' );
00129:  end;
00130:  //-----------------------------------
00131:  procedure TFexcpHdl.FormCreate(Sender: TObject);
00132:  begin
00133:    InitVar;
00134:  end;
00135:  procedure TFexcpHdl.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
00136:  begin
00137:    ReleaseVar;
00138:    canClose:=true;
```

→

ลิสติงที่ 2 (ต่อ)

```
00139:  end;
00140:  procedure TFexcpHdl.Button1Click(Sender: TObject);
00141:  begin
00142:    NoException;
00143:  end;
00144:  procedure TFexcpHdl.Button5Click(Sender: TObject);
00145:  begin
00146:    ConditionProtect;
00147:  end;
00148:  procedure TFexcpHdl.Button2Click(Sender: TObject);
00149:  begin
00150:    CleanUpCode;
00151:  end;
00152:  procedure TFexcpHdl.Button3Click(Sender: TObject);
00153:  begin
00154:    HandleException;
00155:  end;
00156:  procedure TFexcpHdl.Button4Click(Sender: TObject);
00157:  begin
00158:    RaiseYourException;
00159:  end;
00160:  end.
```

Handling the execution : แสดงการทำงานแบบควบคุมการเกิดความผิดพลาด

Edit1 Text box + [Ok] : แสดงการใช้ User-Exception Handler

โปรแกรมนี้จะแสดงรูปแบบการทำงานในระบบต่าง ๆ ตามปุ่มที่กำหนดไว้ เมื่อรันโปรแกรมนี้ขอให้สังเกตการแสดงข้อความใน Memo1 เนื่องจากผมใช้ Memo1 แสดงลำดับการทำงานของคำสั่งต่าง ๆ ที่เรียกใช้ในโปรแกรม สำหรับกิจกรรมที่สำคัญของโปรแกรมมีดังต่อไปนี้ครับ

กิจกรรม NoException: แสดงการเขียนโปรแกรมในลักษณะไม่ป้องกันความผิดพลาดที่อาจเกิดขึ้น

กิจกรรม ConditionProtect: แสดงการเขียนโปรแกรมแบบป้องกันความผิดพลาดโดยใช้วิธีการตรวจสอบเงื่อนไข (Conditional Test)

กิจกรรม CleanUpCode: แสดงการควบคุมการเกิดข้อผิดพลาดแบบ Excuting cleanup code (ดังที่กล่าวแล้วข้างต้น)

กิจกรรม HandleException: แสดงการควบคุมการเกิดข้อผิดพลาดแบบยกเลิกข้อผิดพลาด (Handling the exception)

กิจกรรม RaiseYourException: แสดงการสร้าง Exception Class ขึ้นใหม่

ข้อคิด

ผมสร้างกิจกรรม ConditionProtect เป็นการเขียนโปรแกรมแบบป้องกันความผิดพลาดที่นักพัฒนาระบบส่วนใหญ่นิยมใช้กัน (ตัวผมด้วย) เพื่อให้เปรียบเทียบหาข้อแตกต่าง และสิ่งที่น่าสนใจของการใช้ Exception Handling กับระบบของเรา เป็นการบ้าน ซึ่งการรันภายใต้โปรแกรมนี้อาจจะไม่เห็นข้อแตกต่างมากนัก แต่ในการพัฒนาระบบจริง ๆ แล้วระบบจะซับซ้อนมากกว่านี้ และกิจกรรมอื่น ๆ ที่มีส่วนป้องกันเช่นเดียวกับกิจกรรม ConditionProtect คงไม่ได้ทำงานเพียงแค่ครั้งเดียว ขอฝากไว้ครับ

ฉบับนี้ผมค่อนข้างเน้นเรื่องการพัฒนาโปรแกรมให้มีเสถียรภาพและการใช้เครื่องมือที่สามารถช่วยเหลือในการพัฒนาโปรแกรมให้รวดเร็วยิ่งขึ้น เนื่องจากผมคิดว่าเป็นสิ่งที่นักพัฒนาระบบหลาย ๆ ท่านสามารถนำไปประยุกต์ใช้งานได้เป็นอย่างดี เนื้อหาที่กล่าวถึงอาจจะไม่ค่อยละเอียดนัก สำหรับผู้ที่ต้องการค้นคว้ารายละเอียดเพิ่มเติมสามารถค้นหาได้จาก Help ของ Delphi 3.0 ในหัวข้อ Exceptions

สำหรับฉบับนี้คงต้องกล่าวคำอำลาเพียงเท่านี้ ฉบับหน้าพบกันใหม่ สวัสดีครับ